Smart Multi-Task Scheduling for OpenCL Programs on CPU/GPU Heterogeneous Platforms

Yuan Wen
School of Informatics
The University of Edinburgh
yuan.wen@ed.ac.uk

Zheng Wang
School of Computing and Communications
Lancaster University
z.wang@lancaster.ac.uk

Michael F.P. O'Boyle
School of Informatics
The University of Edinburgh
mob@inf.ed.ac.uk

***Abstract*—Heterogeneous systems consisting of multiple CPUs and GPUs are increasingly attractive as platforms for high performance computing. Such platforms are usually programmed using OpenCL which provides program portability by allowing the same program to execute on different types of device. As such systems become more mainstream, they will move from application dedicated devices to platforms that need to support multiple concurrent user applications. Here there is a need to determine when and where to map different applications so as to best utilize the available heterogeneous hardware resources. In this paper, we present an efficient OpenCL task scheduling scheme which schedules multiple kernels from multiple programs on CPU/GPU heterogeneous platforms. It does this by determining at runtime which kernels are likely to best utilize a device. We show that speedup is a good scheduling priority function and develop a novel model that predicts a kernel's speedup based on its static code structure. Our scheduler uses this prediction and runtime input data size to prioritize and schedule tasks. This technique is applied to a large set of concurrent OpenCL kernels. We evaluated our approach for system throughput and average turn-around time against competitive techniques on two different platforms: a Core i7/Nvidia GTX590 and a Core i7/AMD Tahiti 7970 platforms. For system throughput, we achieve, on average, a 1.21x and 1.25x improvement over the best competitors on the NVIDIA and AMD platforms respectively. Our approach reduces the turnaround time, on average, by at least 1.5x and 1.2x on the NVIDIA and AMD platforms respectively, when compared to alternative approaches.**

***Keywords*-GPU; OpenCL; task scheduling; machine learning;**

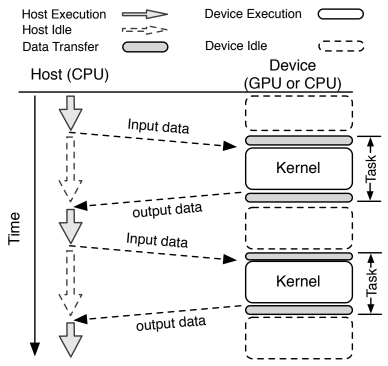
I. Introduction

We now live in the parallel manycore era. Due to power-density constraints, increased single processor performance via ever-increasing clock frequency is no longer possible. This move to parallel system has been mirrored by the growing use of specialised accelerators such as GPUs. Heterogeneous systems consisting of multiple CPUs and GPUs are increasingly attractive as they provide cost-effective, energy-efficient high performance computing

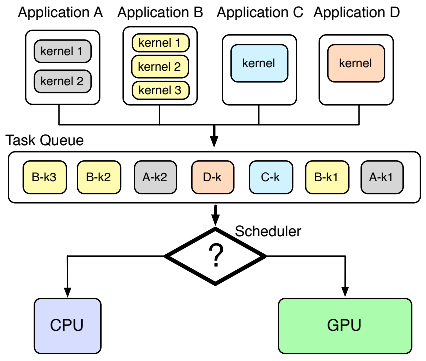
OpenCL [1] has emerged as a standard which provides program portability by allowing the same program to execute on different types of device. Although it provides portable functionality, its performance will vary drastically across different components of the heterogeneous system. Now, as such systems become more mainstream, they will move from application dedicated devices to platforms that need to support multiple concurrent user applications. Performance variability that may be manageable when the GPU is used as a dedicated acceleration device by a single application poses a problem for concurrent users. Here there is a need to determine when and where to map different applications to best utilise the available hardware resources.

In this paper, we address the problem of how to schedule multiple OpenCL applications on a CPU+GPU platform. Although scheduling is a much studied subject [2], [3], [4], [5], [6], [7], heterogeneous scheduling is made more complex by the different execution times an application will experience on different devices [8], [9], [10]. Furthermore, while one application may experience significant performance improvement when moving from a manycore CPU to a GPU, another may experience a slow down. Given a set of application tasks to schedule, it is only possible to determine the best allocation of tasks to devices and their schedule if their execution time is known at schedule time. While this may be possible in certain embedded systems, it is not the case in general purpose systems when the job mix is not known ahead of time. Furthermore, the best schedule can vary depending on optimization criteria; maximizing system throughput may be at the expense of average turnaround time.

We develop a novel scheduling approach which determines at runtime which applications are likely to best utilize a device. We show that speedup is a good heuristic for heterogeneous throughput and develop a novel predictor that determines an application's speedup based on static code structure. However, a speedup alone based priority heuristic would favour small jobs with high speedup over longer jobs with more modest speedup. Our scheduler therefore combines speedup prediction and runtime input data size as factors in considering scheduling priority. This technique is applied to a large set of concurrent programs and evaluated for two distinct metrics: system throughput and average normalized turnaround time. We compare our scheduling approach against FluidiCL[8], a state-of-the-art kernel split

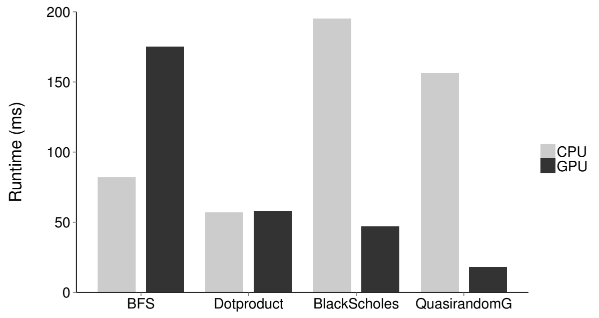


(a) OpenCL tasks

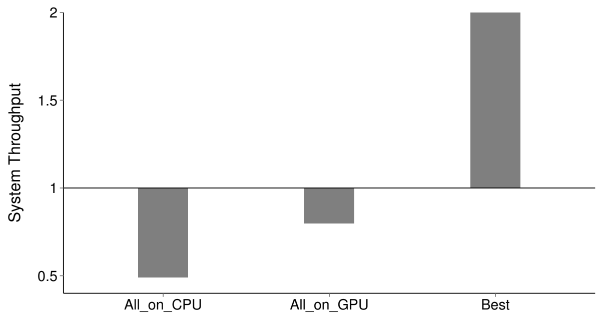


(b) A scheduling scenario

Figure 1. Multi-task scheduling on CPU/GPU heterogeneous systems.



(a) Program Runtime



(b) Scheduling performance

Figure 2. Task scheduling on heterogeneous systems is challenging – the best scheduling depends on the mix of application tasks and executing on the GPU may not be the best strategy.

mapping scheme in which both CPU and GPU are fully used. Our approach shows significant performance improvement, for both metrics, over all other approaches.

The paper makes the following contributions:

- Develops a speedup classifier for OpenCL kernels
- Presents a speedup based scheduling heuristic for heterogeneous platforms
- Demonstrates significant improvement for throughput and turnaround time against existing scheme
- Provides a detailed limit study for OpenCL scheduling

II. Background

This work is concerned with the scheduling of multiple OpenCL kernel tasks on a CPU/GPU based heterogeneous platform. A kernel task is referred to as an OpenCL kernel at runtime, which includes computation and associate CPU-GPU communications. This concept is depicted in Figure 1(a). Tasks might belong to one or more than one OpenCL programs. Note that in this paper we do not split the work of a single kernel across devices.

A typical scenario of OpenCL task scheduling is illustrated in Figure 1(b). Here we have a task queue that is managed by a runtime scheduler. In this example, the task queue contains several OpenCL tasks submitted by four OpenCL programs, where each task can run on both the CPU and the GPU. It is therefore the runtime scheduler's responsibility to decide which device to use to run a particular task that can lead to the best overall performance (e.g. throughtput or turnaround time). This paper aims to develop a portable approach for efficient OpenCL multi-task scheduling and our goal is to maximize the system throughput without significantly increasing the average application turnaround time. The next section provides an example showing that scheduling program task on CPU/GPU based heterogeneous systems is non-trivial.

III. Motivation Example

Consider a scenario of scheduling four OpenCL tasks (kernels) from four OpenCL programs (`bfs`, `BlackScholes`, `Dotproduct`, `QuasirandomG`) on a CPU/GPU heterogeneous system. Figure 2 shows the runtime of each individual kernel when it runs on the GPU or the CPU. As can be seen from this figure, kernel (`bfs`) is long running on the GPU and the other two (`BlackScholes` and `QuasirandomG`) enjoy significant improvement on the GPU. Conversely, the shortest running kernel on the CPU, `Dotproduct`, shows no improvement when scheduled to the GPU.

We now consider how different scheduling policies allocate these tasks to a CPU/GPU platform and investigate their resulting performance. The first one is a greedy first

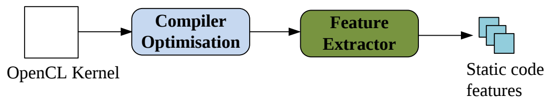


Figure 3. Static code features extraction.

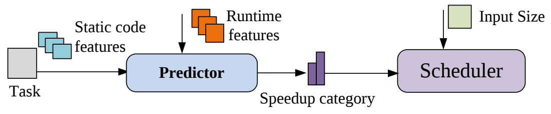


Figure 4. Runtime task speedup prediction

come first served policy which allocates tasks to whichever device is available (`FCFS`). The second policy is to execute the tasks only on the many-core CPU in a FIFO manner (`All_on_CPU`). The third policy runs all tasks on the GPU in a FIFO manner (`All_on_GPU`). Finally we consider the best possible schedule if we were to know the program execution time ahead of time (`Best`). This is impossible in practice but serves as a useful goal for performance.

Figure 2 (b) shows the resulting throughput performance. Here we use `FCFS` as our baseline of 1.00 and show the other policies' relative speedup. The `All_on_CPU` scheme is obviously a poor scheme as it only utilizes the CPU. The `All_on_GPU` policy is more effective, but still not about to give performance improvement over `FCFS`. The `Best` schedule, however, achieves a speedup of 2x, a significant improvement over the other schemes. Clearly, there is significant room for performance improvement for the policies when compared to the best performance.

This example demonstrates that scheduling policy is critical to system throughput. A good policy depends on whether each individual task can benefit from the GPU execution and how long running the task is. If we know this information before scheduling the tasks, we can then determine efficiently which device to use to run each individual task. What we need is a technique that can predict the GPU speedup of any given OpenCL kernels and estimate the running time of a task. The remainder of the paper describes how to predict OpenCL kernels speedup and use these predicted speedups together with input sizes as a guide to schedule tasks across the CPU and GPU.

IV. Overall Scheme

Although knowledge of the execution time of each task is needed for optimal scheduling, accurately determining the execution time of a unseen program is undecidable [11], [12]. Our approach is to use the predicted speedup of an OpenCL kernel when it is to be executed on the GPU as part of the guide to its scheduling priority. High speedup kernel tasks are scheduled to the GPU, lower speedup ones to the CPU.

Determining the potential speedup of a kernel is non-trivial, so we consider a simpler classification problem. We classify programs into two categories *high*, and *low* speedups and use these classification to assign task priority. Accurately classifying programs in this way relies on the structure of the program and the input data size. Although we have access to the code before execution, the input data size will only be known at runtime. As OpenCL is just-in-time (JIT) compiled, we consider code and input size at the same time.

Figures 3 and 4 illustrate our 2-part approach. The compiler extracts static code features from the abstract syntax tree for each OpenCL kernel. These features are then combined with runtime data information to predict which speedup category (high or low) this kernel belongs to when running it on the GPU. The prediction is achieved by way of a machine learning model applied to the OpenCL kernel when compiled by the JIT compiler. The prediction, i.e. the speedup category of the input program for a given input, is used by the runtime scheduler to determine which device to use for each individual task.

At the heart of this approach is a speedup category predictor. In the next section, we will describe how a machine learning based classifier can be built to predict the speedup category of any *unseen* OpenCL programs.

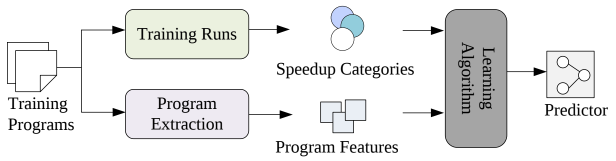


Figure 5. The process of training a machine learning predictor with training examples.

V. Predictive Modeling

Our predictive model for speedup category prediction is a support vector machine. The input to the model is a set of features that describes the input OpenCL kernel. Its output is classification that indicates whether the input kernel is a high-speedup or low-speedup kernel.

A. Building the Predictor

Our predictor is built offline using training programs. The built model can then be used within a OpenCL task scheduler.

Figure 5 depicts the process of training a machine learning model using training programs. The training process involves the collection of training data which is used to fit the model to the problem at hand. In our case we use a set of OpenCL programs that are both executed on the CPU and the GPU to measure the speedup of the GPU execution for each individual kernel over the CPU. Depending on the speedup, each kernel will be labelled as either a high-speedup or a low-speedup category. In this work, an OpenCL kernel will be labelled as high-speedup if the measured GPU-speedup is larger than a certain threshold. Otherwise, it will be labelled as low-speedup. This threshold value was determined experimentally, which is set to 4 in this work.

We also extract features for each kernel as described in the following section. The features together with the speedup

Table I
PROGRAM FEATURES

Static features	#instructions	#load/stores
	#blocks	#br/condbranches
	#mathFunctions	#vector operations
	#int operations	#float operations
	#control instruction	#logic operations
	#barriers	#atomic operations
Runtime features	local_work_size	global_work_size
	input size	output size

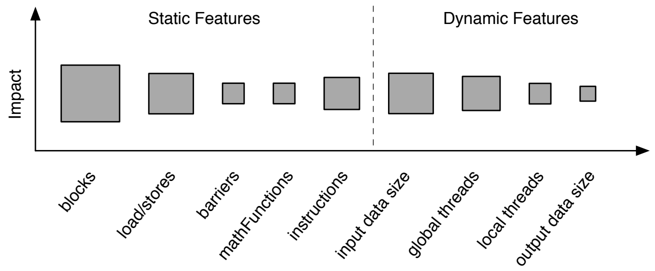


Figure 6. Importance of program features. The larger the box, the more important a feature is for the prediction accuracy.

category for each program from the training data are used to build the model. Since training is only performed once at the factory, it is a *one-off cost*. In our case the overall training process takes less than a day on a single machine.

Predictive Model: Our model is a support vector machines classifier [13]. We use the Radial basis function kernel, which is able to model both linear and non-linear classification problems. We chose SVM as it gives better prediction accuracy when compared to other models (i.e. K-nearest neighbour and decision trees) in our case.

B. Program Features

Our predictor uses program features to characterize an OpenCL program. We use both static code features, such as the number of instructions, and parallel runtime parameters, such the number of work items. All the static and runtime features are listed in Table I.

Static code features are extracted from the abstract syntax tree of the OpenCL kernel at the time the program is compiled by the OpenCL just-in-time compiler. The feature extraction tool is based on Clang and LLVM [14]. At compile time, we extract information about the number and type of operations.

Besides static code features, we also use *parallel runtime features* to characterize the dynamic behavior which is often associated with the program input. The `local_work_size` and `global_work_size` indicate the maximum number of current threads, which are useful for determining the amount of parallelism available. The memory transfer represents the communication overhead between the multi-core CPU and the GPU, which can have a significant impact on the GPU speedup. The `input/output size` is estimated by calculating the number of bytes to be transferred between the host CPU and the GPU.

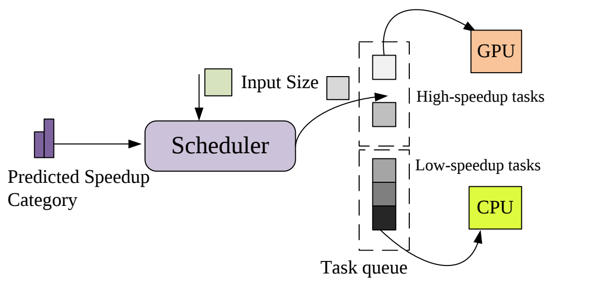


Figure 7. New arriving tasks will be inserted into an appropriate position of the task queue based on their predicted speedup categories and input sizes. Tasks from each end of the queue will be dispatched onto the GPU and CPU respectively.

Our predictor is trained with all static features and dynamic features which are shown in Table I. Features which contribute least to the prediction is filtered out by our training process. In our experiment, we only select five static features and four dynamic features. Using fewer carefully selected training features is able to shrink predictor training time without suffering a lost in accuracy. In this paper, our selected features is shown in Figure 6. The overall contribution of each selected feature in our training process is also shown in Figure 6.

VI. RUNTIME TASK SCHEDULING

Newly arriving OpenCL kernels are inserted into a task queue from which kernel tasks are dequeued and scheduled to either the CPU or GPU when the devices are available as shown in Figure 7. The queue is sorted based on the predicted speedup category and program input size. High-speedup kernel tasks are dequeued from one end and scheduled to the GPU, low speedup task are dequeued from the other end and scheduled to the CPU. Tasks will be firstly grouped according to their speedup category where tasks with the same speedup category will be placed together. Those tasks will then be sorted according to the input size in a way that those tasks with relatively smaller input sizes will be placed towards the end of the queue where the CPU will take tasks from. This is because based on our observation tasks with a large input size often correlates with long execution time. We always prefer to schedule tasks that have long runtime but can enjoy GPU execution with a high-speedup onto the GPU.

Since tasks are dequeued from both sides of the task queue, this dequeue process will meet at somewhere in the middle of the queue. The last task of the queue will be replicated and mapped to both the CPU and the GPU. The last task will be first scheduled to an available device and when the other device becomes idle the scheduler will map a duplicated copy to this device. The scheduler then waits for one of the tasks to complete and kills the outstanding duplicate.

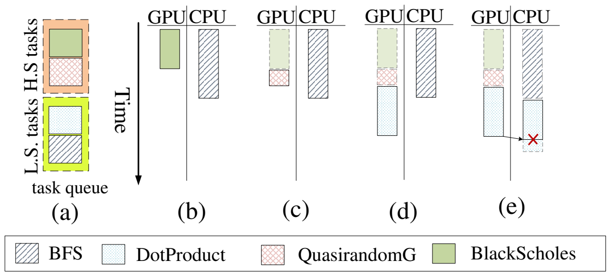


Figure 8. The layout of the task queue (a) and the scheduling process (b) (e) for the example shown in Figure 2 using our machine learning based scheduler. In this case, our approach achieves the optimal throughput.

Scheduling Example: Figure 8 shows how the four tasks, `bfs`, `Dotproduct`, `BlackScholes`, `QuasirandomG`, presented in Figure 2 are scheduled by our scheduler. Our predictor takes the feature values for each OpenCL kernel task and predicts to use the CPU or GPU for scheduling. For instance `BlackScholes` is classified by our predictor as a high-speedup category, it is scheduled on the GPU. `bfs` has a set of different feature values, which is classified by the predictor as low-speedup task, it is scheduled on the CPU. Both `BlackScholes` and `QuasirandomG` are classified as high-speedup tasks and the other two are classified as low-speedup tasks. Based on their speedup categories and input sizes, the tasks are sorted in the task queue as shown in Figure 8 (a). If we assume both the GPU and CPU are available upon the time those tasks arrive, this will result in a scheduling plan as depicted in Figures 8 (b) - (e) over time. For this example, our scheduler gives the best throughput performance.

VII. Alternative Policies

A. Alternative Scheduling Policies

We compare our approach against four different strategies:

- **`All_on_CPU`**. Using this scheme, tasks are dispatched to the shared CPU in the arriving order.
- **`All_on_GPU`**. Using this scheme, tasks are dispatched to the shared GPU in the arriving order.
- **`FCFS`**. This is a first come first served approach. Using this scheme, application tasks will be put into the task queue in the order as they arrive. Then tasks will be dispatched to any available computing device (either the GPU or the CPU).
- **`Input size guided`**. In the task queue, tasks are sorted based on the amount of bytes needed to be transferred from the CPU to the GPU. With this scheme, the GPU always gets a task that has the largest input and the CPU always gets a task with the smallest input.
- **`Work item guided`**. In the task queue, tasks are sorted according to the number of global work items of the kernel. Using this scheme, the GPU always gets a task that has the largest number of work items while the CPU always gets a task with the smallest number of work items.

Table II
Hardware platform

	Intel CPU	NVIDIA GPU	AMD GPU
Model	Core i7 2600K	GeForce GTX 590	Radeon HD7970
Core Clock	3.4 GHz	1215 MHz	1000 MHz
Core Count	4 (8 w/ HT)	1024	2048
Memory	8 GB	3 GB	3GB
Memory Bandwidth	21GB	327 GB	288 GB

There are some alternative scheduling schemes, such as the shortest-job-first scheme [15], which all require to know the task execution time ahead of time. Since our experimental settings assume this information is not available to the scheduler, those approaches cannot provide a fair comparison and hence are not included. Round Robin is another widely used task scheduling scheme. However, because the current GPU implementation does not support context switch or preemption, this is not available for comparison either.

B. Partitioning OpenCL Kernels across Devices

The FluidiCL [8] runtime utilizes both the multi-core CPU and the many-core GPU to concurrently execute a single OpenCL kernel. In this way, the CPU executes part of the kernel, starting from the upper end of the working space, while the GPU executes the whole kernel, but starting from the lower end of the working space. When the GPU reaches a work-group that has already been executed by the CPU, the whole kernel execution is considered to have been completed and the results will be merged. However, this scheme can only apply to one single OpenCL kernel. As a result, kernels from multiple applications will have to be executed sequentially. Furthermore, distributing work items between the CPU and GPU requires synchronization and communications between the two devices, which can incur significant runtime overhead. We compare our approach against FluidiCL in Section IX-B

VIII. Experiment Setup

This section describes our experimental setup and the evaluation methodology used in the remainder of the paper.

A. Platform and Benchmarks

Platform and Software Tools: We evaluate our approach on two CPU-GPU systems: both use an Intel Core i7 4-core CPU. One system contains an NVIDIA GeForce GTX 590 GPU, the second an AMD HD 7970 GPU. Both run with the OpenSUSE 12.3 Linux. Our compiler is GCC 4.7.2 with -O3 as the compiler option. We use the NVIDIA CUDA Toolkit 3.1 which has an OpenCL just in time compiler. Details of the hardware platforms are shown in Table II.

Benchmarks: We used 35 different benchmarks from three mainstream OpenCL benchmark suites: the NVIDIA OpenCL SDK v4.2, the AMD SDK v2.8 and the Parboil OpenCL benchmark suite v2.5. In the experiments, we ran

Table III
BENCHMARKS AND INPUT SIZES

Suite	Benchmarks	Input Size	Benchmark	Input Size
NVIDIA	BlackScholes	12K - 12M	ConvolutionSeparable	1.6M - 420M
	DXTCompression	17M - 604M	DotProduct	41M - 654M
	FDTD3d	452M	HiddenMarkovModel	69M
	Histogram	67M - 268M	MatrixMul	63M
Parboil	BFS	64M	Cutcp	3M - 36M
	Sgemm	192K - 12M	Spmv	49K - 31M
AMD	BinarySearch	2K	BinomialOption	3K
	BitonicSort	16K - 65K	BlackScholes	1M - 4M
	BlackScholesDP	2M - 5M	DCT	16K - 16M
	DwtHaar1D	4K - 65K	FastWalshTransform	4K - 131K
	FloydWarshall	262K	Histogram	4M-1G
	MatrixMultiplication	16K - 1M	MatrixTranspose	16K - 67M
	PrefixSum	2K - 16K	QuasiRandomSequence	1K
	Reduction	8K	ScanLargeArrays	4K - 65K
	SimpleConvolution	16K		
Ploybench	ATAX	1G	BICG	1G
	CORR	50M	GESUMMV	1G
	SYR2K	50M	SYRK	33M

each benchmark with a range of different inputs. The list of benchmarks and inputs is shown in Table III.

B. Runtime Scenarios

Our evaluation setting consists of multiple runtime scenarios with 49 different task mixes where each task mix contains 2 to 50 OpenCL kernels (tasks). The task mixes are grouped into three task groups with different numbers of tasks: *small*, *medium*, and *large*. We consider a task group to be small, medium and large if it contains less than 10, 10-20, or more than 20 (upto 50) kernels respectively. For each task mix, we tried up to 125 different task combinations with different OpenCL kernels and input sizes. We report the average performance per task group as the *geometric mean* across all combinations. The OpenCL applications of each task group were randomly selected from the list of benchmarks given in Table III. Moreover, in the experiments we replayed each scheduling decision 10 times and calculated the average performance of each decision to reduce the impact of jitter. Finally, we assumed all tasks arrive at the same time and have the same priority.

C. Performance Evaluation

Performance Metrics: To evaluate our approach, we used two metrics, *system throughput*, a system oriented metric, and *turnaround time*, a user oriented metric. Those two metrics have widely been used to evaluate the performance of a scheduler in a multi-tasking environment [2], [4]. Our goal is to maximize the system throughput which in general leads to favourable turnaround time results. The definitions of the two metrics are given as below.

System throughput (STP) is a *higher is better* metric. It describes the number of tasks completed per unit time. This is calculated by using the `FCFS` scheme as a baseline of 1.0, showing the relative speedup of other scheduling policies. It is defined as

$$STP = \frac{\sum T^i_{FCFS}}{\max(\sum T^m_{cpu}, \sum T^n_{gpu})} \tag{1}$$

where T^i_{FCFS} is the execution time given by `FCFS`, and T^i_{cpu} and T^i_{gpu} are the execution time by running task T^i on the CPU and the GPU respectively.

Average normalized turnaround time (ANTT) is a *smaller is better* performance metric. It quantifies the time between a task is created and its completion, indicating the average user-perceived delay in multi-tasking environment compared to running a single task on the system. In the experiments, the turnaround time is normalized to the `FCFS` scheme. ANTT is defined as:

$$ANTT = \frac{1}{n}\sum_{i=1}^{n}\frac{T^i_{sch}}{T^i_{FCFS}} \tag{2}$$

where T^i_{FCFS} and T^i_{sch} are the time between task T^i is created and its completion using `FCFS` and an alternative scheduling policy respectively.

Predictive Modeling Evaluation We use *leave-one-out cross-validation* to train and evaluate our predictive modeling based scheduler. This means we remove the target OpenCL programs to be predicted from the training program set, collecting training examples without the target programs to be presented, and then learning a model from the training examples. It is a standard evaluation methodology, providing an estimate of the generalization ability of a machine learning model in predicting unseen programs.

IX. RESULTS

In this section we present the experimental results of our approach. The baseline is `FCFS`. Our goal is to maximize the STP and minimizing the ANTT, so that the system can finish as many tasks as possible within per time unit and at the same time reducing the turnaround time.

A. Overall Results

STP: As can be seen from Figures 9 (a) and 10 (a), our approach consistently outperforms the baseline for this higher is better metric. As the number of tasks to be scheduled increases, we see an overall increase in the STP. The baseline `FCFS` scheme performs well for a small task group and all other alternative approaches give slowdown performance except for our approach where a 1.1x improvement is observed on both platforms. The improvement of our approach increases to an average STP of 1.4 for a large task group. This is not an unexpected result as the `FCFS` simply allocates a task to the first available device without considering which the most appropriate computing device is. This scheme may work well for a small task group as the number of available scheduling options is small, but is unlikely to achieve good performance as the number of tasks to be scheduled increases where a large number of scheduling options is opened up. By assigning high-speedup tasks to the GPU and low-speedup ones to the CPU, our approach can make effective use of both GPU and CPU and achieves higher throughput. On average, our approach achieves a throughput of 1.25 across all task group sizes. This significantly outperforms other approaches which all fail to improve the STP.

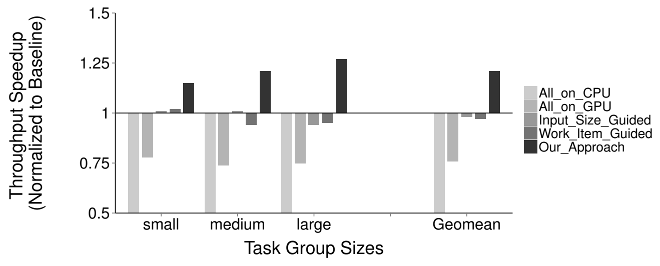
ANTT: Figures 9 (b) and 10 (b) show the achieved ANTT, a lower is better metric, on the NVIDIA and AMD platforms respectively. As can be seen from the diagrams, our approach not only improves throughput and but also the ANTT. Our approach constantly outperforms the baseline and has a lower ANTT as the number of tasks to be scheduled increases. The ANTT given by our approach is 0.58 and 0.8 for a small task group on the NVIDIA and the AMD platforms respectively, which is further reduced to less than 0.6 for a large task group. Similar to the STP results, our ANTT performance is improved as the number of tasks to be scheduled increases where the number of available scheduling options increases. Overall, our approach performs well with an average ANTT of 0.56 and 0.65 on the NVIDIA and the AMD platforms respectively. On average, our approach outperforms all other approaches by reducing the turnaround time by at least 1.5x and 1.2x on the NVIDIA and the AMD platforms respectively.

Summary: Our approach constantly outperforms all alternative approaches for two performance metrics: STP and ANTT. The advantage of our approach is largely attribute to its capability to predict the potential speedup category of each kernel task. Without this information, the alternative schemes may inappropriately assign tasks onto the GPU, which may not be able to benefit from the GPU execution. This leads to the poor GPU utilization and overall poor scheduling performance.

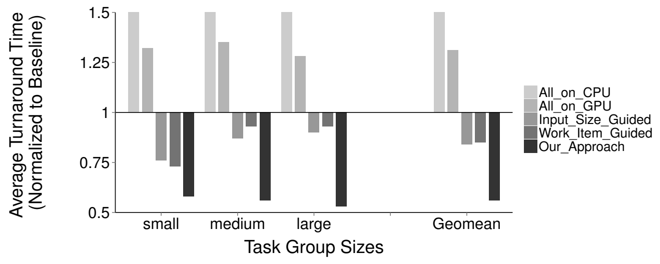
B. Comparison to State-of-the-Art

Figure 11 (a) compares the STP improvement achieved by our approach against FluidiCL on the NVIDIA platform. The performance of FluidiCL is disappointing when scheduling multiple OpenCL tasks. It gives an average slowdown of 0.93 over `FCFS`. Only for small task groups, by partitioning the work of OpenCL kernels across the CPU and the GPU, FluidiCL is able to achieve a modest speedup (1.03x) over `FCFS`.

Using FluidiCL, a faster computing device will eventually execute a large portion of the work. For all the OpenCL kernels we used in the experiments, there is always one computing device (either the CPU or the GPU) clearly outperforms the other. As a result, the use of an additional computing device rarely accelerates the computation of a single kernel. Also, we observed that FluidiCL often introduces expensive synchronization and communication overhead between the two computing devices for distributing work and merging results, leading to overall slowdown performance. Unlike FluidiCL, our approach avoids such synchronization and communication overhead, giving constantly better STP performance over FluidiCL. On average, our scheme improves the STP by 1.23x compared to FluidiCL. Furthermore, our approach is able to minimize the average turn around time by 1.96x over FluidiCL (0.55 vs 1.08 in Figure 11 (b)).

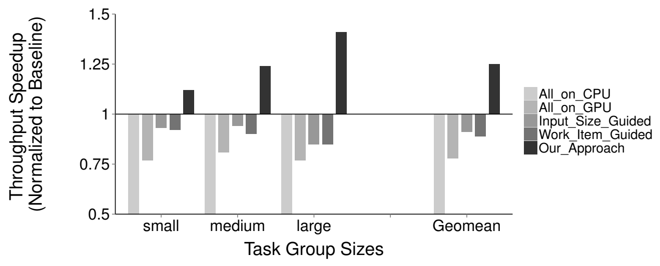


(a) System throughput on the Nvidia Platform

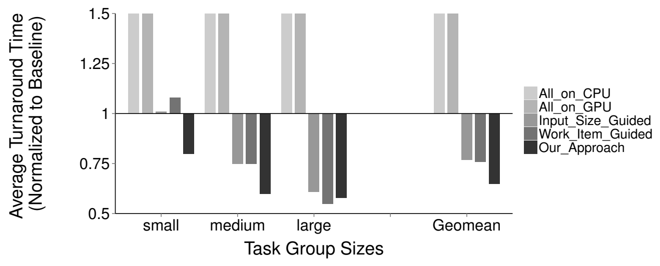


(b) Normalize average turnaround time on the Nvidia Platform

Figure 9. The achieved STP (a) and ANTT (b) on the Nvidia GPU platform. Our approach achieves, on average, a 21% and 56% of improvement over the baseline (`FCFS`) for the STP and ANTT metrics respectively.

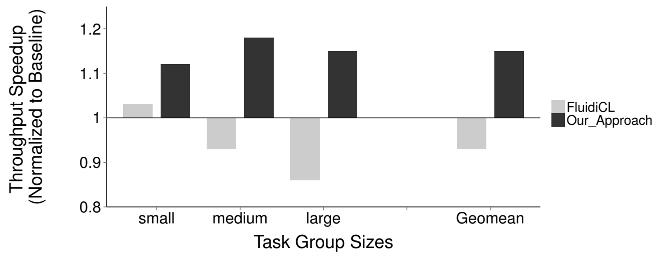


(a) System throughput on the AMD Platform

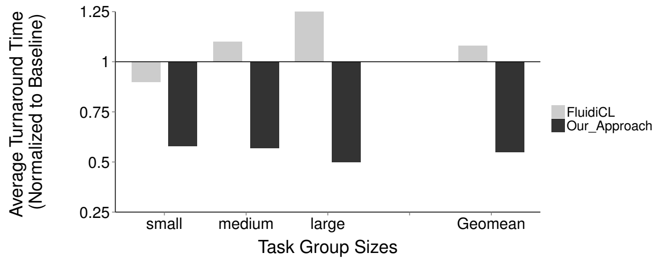


(b) Normalize average turnaround time on the AMD Platform

Figure 10. The achieved STP (a) and ANNT (b) on the AMD GPU platform. Our approach achieves, on average, a 25% and 65% improvement over baseline (`FCFS`) for the STP and ANTT metrics respectively.



(a) System throughput of our approach vs FluidiCL



(b) Normalize average turnaround time of our approach vs FluidiCL

Figure 11. Our approach constantly outperforms FluidiCL with an average improvement of STP (a) (1.15x vs 0.93) and ANTT (b) (0.55 vs 1.08) on the NVIDIA platform.

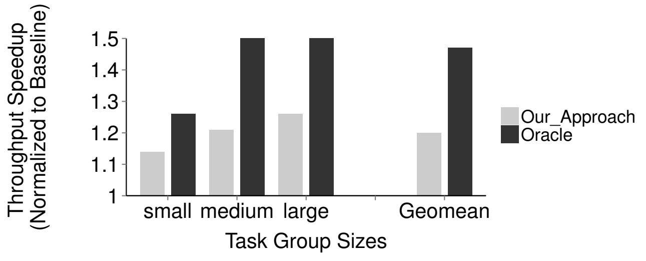


Figure 12. Comparison to the best available STP. Our approach achieves 43% of the best available performance on the NVIDIA platform.

X. Analysis

A. Limit Study

1) Best Available Performance: Although our scheme performs well compared to alternative approaches, it is useful to know whether there is any further room for improvement. It may be the case that the competitive schemes are very poor and that a smarter scheme could perform significantly better. In Figure 12, we compare our scheme against the best available STP performance on the NVIDIA platform. This is obtained by exhaustively trying all possible scheduling options. This `best STP` scheduling is unrealistic in practice, but provides a useful upper bound.

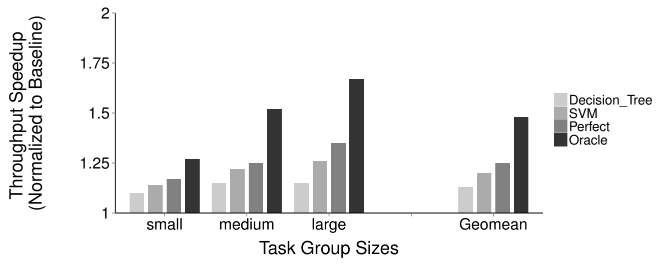
When there is a small number of kernel tasks to schedule, our approach gives nearly optimal performance. When the number of kernel tasks is large, there is room for improvement. The `best STP` schedule is able to improve performance by 50% for the STP. This is because speedup is only a proxy for execution time. Errors in estimating execution time will increase as the number of tasks increase. In fact while we can determine the `best STP` schedule, it is impossible to determine accurately the best ANTT schedule due to combinatorial complexity. While we will never achieve the performance of the `best` schedule as it is undecidable, there is still room for improvement. In the next experiment, we therefore evaluated prediction accuracy and see if this has an impact on performance.

2) Impact of Prediction Accuracy: We would like to know how the prediction accuracy affects the scheduling performance. To do so, we have also considered a decision tree based model and a theoretically `Perfect` predictor which always gives the correct speedup classification (i.e. the prediction accuracy is 100%).

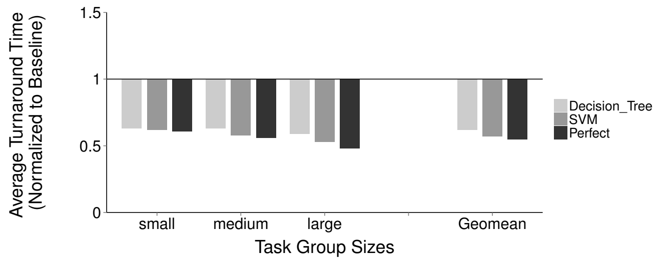
Figure 13 compares the STP and ANTT performance achieved by the three models. As can be seen from the diagram, prediction accuracy has significant impact on the scheduling performance and in fact the more accurate a predictor is the better performance the scheduler has. The SVM model has higher accuracy (87%) than the decision tree model (72%) for STP (Figure 13 (a)). The SVM model therefore gives constantly better results for both evaluation metrics when compared to the decision tree model (1.2x vs 1.13x). A `Perfect` classifier further increases performance to 1.25x. We see a similar pattern for ANTT (Figure 13 (b)). Here the SVM model gives an ANTT of 0.57. The decision tree once again degrades performance to 0.62 while the `Perfect` predictor improves it. Although building a `Perfect` predictor is almost impossible in reality, this experiment result confirms that the scheduling performance can be further improved with a more accurate model. However, by comparing to the best available performance of STP[1] there is still room for performance improvement even for the *perfect* predictor. One reason may be that our current approach only has two speedup categories which essentially is a coarse-grained classification. This hypothesis is confirmed by the experiment described in the next section.

3) Fine-grained Speedup Categorization: Currently, each kernel task is classified into two speedup categories. We want to know whether a finer-grained classification could improve scheduling performance. To do so, we first break down the number of speedup categories to create 3-7 categories; we then classify the speedup of each kernel task using the actually measured speedup. Figure 14 shows the STP performance using different category granularities. As can be seen from this figure, a finer-grained classification in general leads to better STP performance. This shows that our approach can be further improved using finer-grained classification and this is our future work.

[1]Given the extremely large combinatorial scheduling option available, it is infeasible to find the best ANTT performance. Therefore, we do not present the best ANTT performance.



(a) System Throughput



(b) Normalized Average Turnaround Time

Figure 13. The STP (a) and ANTT (b) achieved by different predictive models. The more accurate the model is the better the scheduling performance is.

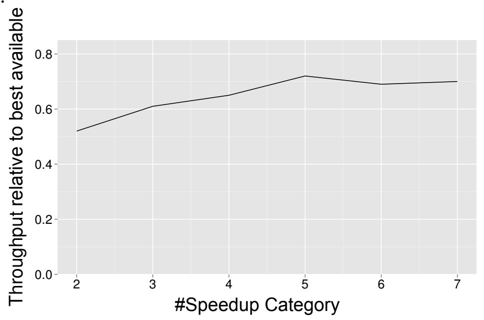


Figure 14. STP performance tends to improve with a finer-grained speedup category classification.

B. Overhead

Our predictive model is trained offline with training examples. In this work, collecting the training examples took less than a day using a single machine, which has no impact on runtime cost. The overhead of using the trained model includes extracting program features and making predictions. This overhead is negligible (approximate 10ms in total), which has been included in all experimental results.

XI. Related Work

Programming Frameworks for GPUs: As GPUs become increasingly available, there have been correspondingly programming models [16], [17] and compiler tools [18] proposed for GPU programming. These approaches provide APIs to develop GPU applications. All these approaches implicitly assume the GPU execution gives the best performance and the program is often only tuned for the GPU.

Program Mapping for GPUs: A number of approaches have been proposed to partitioning a GPU program kernels across the CPU and the GPU. The Qilin [10] compiler first uses profile runs to build a regression model for the target program and then uses the built model to predict the optimal loop iteration distribution among the CPUs and GPUs. Grewe *et al* [19], [20] present an OpenCL kernel partition approach which takes GPU contention into account. Other works takes multiple GPUs into consideration [21], [22], [23]. All of these approaches, however, only consider the mapping of a single parallel application or job. None of them consider how to schedule multiple kernel tasks from different OpenCL programs.

Dynamic Task Scheduling: There has been a significant amount of prior work investigating hardware and operating system based approaches to schedule multiple tasks on CPUs. For examples, symbiotic job scheduling tries to find the best mix of jobs [2], [4] on SMT processors; and Parcae is a dynamic tuning framework [24] to improve job co-execution on multi-core CPUs. Other work [25], [26] profiles the program to collect information such as the number of memory load and store operations and uses this information as an indicator to schedule multiple tasks across asymmetric multiprocessors. All those schemes require runtime profiling or searching to determine a scheduling plan, which may incur significant runtime overhead.

Scheduling on CPU/GPU Systems: Examples of prior work on task scheduling on CPU/GPU systems include [27], [28] and [29], which all require the estimated runtime of each task. Since providing accurate execution time estimation can be unrealistic in many cases, those approaches can only applied to a limited classes of applications. Rather than relying on task runtime which is difficult to obtain in practice, our novel approach uses machine learning to predict the speedup category of a given OpenCL kernel based on static program structures and runtime program input that are available to the just-in-time OpenCL compiler. Some other work [30], [31] considers kernel work partitioning across different processors. These approaches are orthogonal to our approach. Some of the most recent work [32], [28] use profiling information to schedule OpenCL tasks. Unlike our approach, these schemes may incur expensive profiling overhead at runtime. StarPU [5] supports the scheduling of a single application on heterogeneous platforms. It relies on a user-provided cost function for scheduling, but how to develop a portable cost function remains an open problem. To support the concurrent execution of multiple StarPU applications with the minimal interference, Hugo *et al* [33] proposed partitioning computing resources into different sets that are managed by different scheduling algorithms. Unlike StarPU which requires the user to provide a cost function for scheduling on each targeting platform, our approach automatically learns a such mapping heuristic using machine learning and is portable across platforms.

Predictive Modeling: Machine learning based predictive modeling has recently been proven to be effective at learning how to optimize programs on an unloaded machine [34]. Recent studies have shown that predictive modeling is effective in optimizing parallel programs [35], selecting micro-architectural parameters [36], and scheduling tasks on homogeneous multi-cores [7]. However, none of the previous research in predictive modeling based program optimization addresses the problem of multi-task scheduling on a heterogeneous platform that consists of different computing devices.

XII. Conclusions

This paper has presented an efficient OpenCL task scheduling scheme which schedules multiple programs across CPUs/GPUs heterogeneous platform. Our scheduler uses a speedup predictor and runtime input data size to schedule tasks. This technique is applied to a large set of concurrent programs where it shows significant performance improvement over all other existing approaches. This work shows that speedup is a good priority function. Future work will investigate improving speedup prediction accuracy using larger training data sets. Significant further improvement is potentially available when using more fine-grained speedup classification.

References

[1] Khronos, "OpenCL: The open standard for parallel programming of heterogeneous systems. http://www.khronos.org/opencl/." [Online]. Available: http://www.khronos.org/opencl/

[2] A. Snavely and D. M. Tullsen, "Symbiotic jobscheduling for a simultaneous multithreaded processor," in *ASPLOS-IX*, 2000.

[3] Y. Zhang, X. S. Hu, and D. Z. Chen, "Task scheduling and voltage selection for energy minimization," in *DAC '02*.

[4] S. Eyerman and L. Eeckhout, "Probabilistic job symbiosis modeling for smt processor scheduling," in *ASPLOS XV*, 2010.

[5] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "Starpu: A unified platform for task scheduling on heterogeneous multicore architectures," *Concurr. Comput. : Pract. Exper.*, vol. 23, no. 2, 2011.

[6] A. K. Singh, M. Shafique, A. Kumar, and J. Henkel, "Mapping on multi/many-core systems: Survey of current and emerging trends," in *DAC '13*.

[7] M. K. Emani, Z. Wang, and M. F. P. O'Boyle, "Smart, adaptive mapping of parallelism in the presence of external workload," in *CGO '13*.

[8] P. Pandit and R. Govindarajan, "Fluidic kernels: Cooperative execution of opencl programs on multiple heterogeneous devices," in *CGO '14*.

[9] E. Sun, D. Schaa, R. Bagley, N. Rubin, and D. Kaeli, "Enabling task-level scheduling on heterogeneous platforms," in *GPGPU '12*.

[10] C.-K. Luk, S. Hong, and H. Kim, "Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in *MICRO 42*, 2009.

[11] S. Hong and H. Kim, "An integrated gpu power and performance model," in *ISCA '10*.

[12] Y. Zhang and J. Owens, "A quantitative performance analysis model for gpu architectures," in *HPAC '11*.

[13] C. M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.

[14] "The LLVM compiler infrastructure project. http://llvm.org/,."

[15] M. L. Pinedo, *Scheduling: Theory, Algorithms, and Systems*, 3rd ed. Springer Publishing Company, Incorporated, 2008.

[16] A. Hormati, M. Samadi, M. Woh, T. N. Mudge, and S. A. Mahlke, "Sponge: portable stream programming on graphics engines," in *ASPLOS '11*.

[17] J. Kim, H. Kim, J. H. Lee, and J. Lee, "Achieving a single compute device image in opencl for multiple gpus," in *PPoPP '11*.

[18] Z. Wang, D. Powel, B. Franke, and M. F. P. O'Boyle, "Exploitation of gpus for the parallelisation of probably parallel legacy code," in *CC '14*.

[19] D. Grewe, Z. Wang, and M. F. P. O'Boyle, "Opencl task partitioning in the presence of gpu contention," in *LCPC '13*.

[20] D. Grewe, Z. Wang, and . M. F. P. O'Boyle, "Portable mapping of data parallel programs to opencl for heterogeneous systems," in *CGO '13*.

[21] H. P. Huynh, A. Hagiescu, W.-F. Wong, and R. S. M. Goh, "Scalable framework for mapping streaming applications onto multi-gpu systems," in *PPoPP '12*.

[22] K. Sajjapongse, X. Wang, and M. Becchi, "A preemption-based runtime to efficiently schedule multi-process applications on heterogeneous clusters with gpus," in *HPDC '13*.

[23] V. T. Ravi, W. Ma, D. Chiu, and G. Agrawal, "Compiler and runtime support for enabling generalized reduction computations on heterogeneous parallel configurations," in *SC '10*.

[24] A. Raman, A. Zaks, J. W. Lee, and D. I. August, "Parcae: a system for flexible parallel execution," in *PLDI '12*.

[25] D. Koufaty, D. Reddy, and S. Hahn, "Bias scheduling in heterogeneous multi-core architectures," in *EuroSys '10*.

[26] T. Li, P. Brett, R. Knauerhase, D. Koufaty, D. Reddy, and S. Hahn, "Operating system support for overlapping-isa heterogeneous multi-core architectures," in *HPCA '10*.

[27] V. Ravi, M. Becchi, G. Agrawal, and S. Chakradhar, "Valuepack: Value-based scheduling framework for cpu-gpu clusters," in *SC '12*, 2012.

[28] V. T. Ravi, M. Becchi, W. Jiang, G. Agrawal, and S. Chakradhar, "Scheduling concurrent applications on a cluster of cpu-gpu nodes," in *CCGRID '12*.

[29] U. Verner, A. Schuster, M. Silberstein, and A. Mendelson, "Scheduling processing of real-time data streams on heterogeneous multi-gpu systems," in *SYSTOR '12*.

[30] F. Song and J. Dongarra, "A scalable framework for heterogeneous gpu-based clusters," in *SPAA '12*.

[31] K. Kofler, I. Grasso, B. Cosenza, and T. Fahringer, "An automatic input-sensitive approach for heterogeneous task partitioning," in *ICS '13*.

[32] G. Teodoro, T. D. R. Hartley, U. Catalyurek, and R. Ferreira, "Run-time optimizations for replicated dataflows on heterogeneous environments," in *HPDC '10*.

[33] A.-E. Hugo, A. Guermouche, P.-A. Wacrenier, and R. Namyst, "Composing multiple starpu applications over heterogeneous machines: A supervised approach," in *IPDPSW '13*.

[34] S. Long and M. F. P. O'Boyle, "Adaptive java optimisation using instance-based learning," in *ICS '04*.

[35] Z. Wang and M. F. P. O'Boyle, "Partitioning streaming parallelism for multi-cores: A machine learning based approach," in *PACT '10*.

[36] M. Zuluaga, A. Krause, P. Milder, and M. Püschel, "Smart design space sampling to predict pareto-optimal solutions," in *LCTES '12*.